Low-level vision

Adrian F. Clark <alien@essex.ac.uk>
CSEE, University of Essex

What is *low-level* vision?

Uses pixel-level information, not things like edges and corners. . . and not machine learning

What do we know about?

- histograms
- convolution and friends

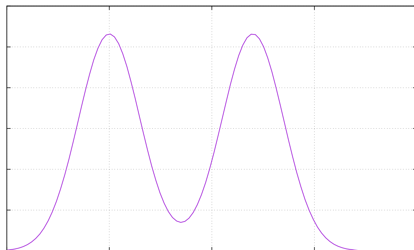
This is enough for us to build a working vision system, as we shall see

The approach

1. Distinguish foreground and background objects
2. Tidy up the objects
3. Describe the objects
4. Interpret these descriptions

Distinguishing foreground and background

This is most easily done using histograms



We want well-separated peaks

I'll describe the process as though the background is dark and the foreground light but it could just as easily be the other way around

Well-separated peaks and standard deviations

If the foreground and background peaks are to be separated, we really want their s.d.s to be minimized

Where do we put our threshold to achieve this?

The best place is between the peaks

Otsu's method

Otsu was the first person to realize that minimizing the s.d. of the peaks was the same as maximizing the distance between them — and that turns out to be easier to do

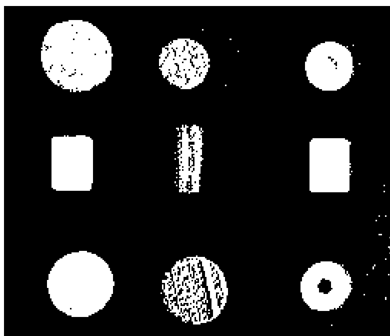
The underlying maths is in the lecture notes; I promise not to test you on it! The algorithm is under a page of Python (or C. . .) and is on p59 of the lecture notes

Otsu's method gives us an *automatic* way of finding a good threshold that splits the background and foreground

It works only for two peaks as described here but can be extended to more

It works best when there are about the same number of pixels in foreground and background

With it, we can *binarize* an image into light and dark regions



Region labelling

Now that we have found distinct regions in the image, it would help if we could assign the same number to all the pixels in each region

This is known as *region labelling*, *connected-component labelling* or *blob labelling*

There are two ways to go about this

Recursive region labelling

This is similar to a flood-fill algorithm in computer graphics

We scan the rows and columns of the image in the usual way until we meet a non-background pixel at position 1, which we mark as a new region

We mark foreground pixel at position 2 as part of the same region and recursively invoke ourself — and then at positions 3 and 4 when it eventually returns

```
#####################      '#' represents background
#####1256789#########
#####34     #########
#####################
```

The recursive invocations continue until we have filled positions 5–9 of the first line; the other lines are filled from positions 3 and 4

This recursive algorithm is compact and elegant — but because it is recursive, it can easily overflow the stack space available to your program

```
HHHHHHHHHH ->         <- SSSSSSSSSSSSS
^                                    ^
0x0000                          0xFFFF
```

The heap `H` (from `malloc`, `new` *etc*) typically starts at low memory and grows upwards while the stack `S` starts at the top of memory and grows downwards

There is usually an artificial limit placed on the maximum stack size (*e.g.*, in Java) to avoid it running into the heap

The bottom line is that the compact, elegant algorithm is poor in practice

Non-recursive region labelling

```
#####################        '#' represents background
#####1      #########
#####2      #########
#####################
```

We scan the rows and columns of the image as in the recursive case but, when we find a new pixel in a region at position 1, we look at the *north* and *west* neighbours: if one of them is already labelled, we use the same label, otherwise we start a new region

The remainder of the region on the same line gets the same label from the *west* neighbour

On the next line, at position 2, we get the same label from the *north* neighbour

There has to be special code for the first line and column

The numbering goes squiffy on U-shaped regions, requiring us maintain a 'label equivalence' table and then re-number some of the regions in a second pass through the image

Together, these make the code quite a bit longer than for the recursive algorithm. . . though it does work for any size image

Some implementations also look at the *north-west* pixel, and that can result in a different set of regions — an example is given in Figure 5.3 of the notes

What we end up with

The result of these algorithms is a set of separate image regions, each of which has a separate label (number) to identify it

```
000000000000000000000000000000
000000001100000000222222222000
000001111111000000222222222000
000011111111100000222222222000
000001111111000000222222222000
000000011100000000000000000000
000000001000033000000000000000
000000000000000000000000000000
```

All we have to do now is find a way of describing each region

Circularity

To determine whether a region is circular, we can find its boundary length (we saw how to do this using morphology) and count the number of pixels to measure its area

We know that $C = 2\pi r$ and $A = \pi r^2$, so we can calculate the *circularity*

$$\frac{C^2}{A} - \frac{4\pi r^2}{\pi r^2} = 4\pi$$

If a feature has a circularity close to 4π, we can say it is circular

Rectangularity

```
                      W
                  <------->
000000000000000000000000000000
000000011000000000222222222000 ^
000001111111000000222222222000 | H
000011111111100000222222222000 |
000001111111000000222222222000 v
000000011100003300000000000000
000000000000000000000000000000
```

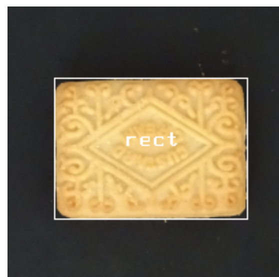
If A is the number of pixels in region 2, W its width and H its height, then we should expect its *rectangularity*

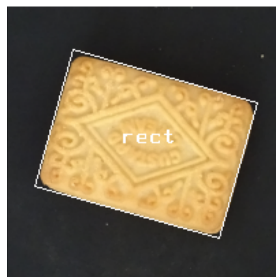
$$\frac{WH}{A} \approx 1$$

Oriented rectangles

One problem that arises with rectangles is that they are not necessarily aligned with the edges of the images

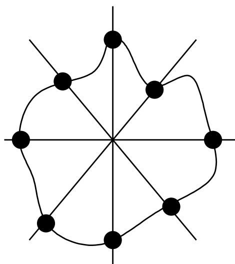
When this is the case, you need to calculate an *oriented bounding box* to find its height and width correctly





More general features

If you are trying to match a shape which is more general than a circle or rectangle, there are approaches such as



Extremal points in different directions can describe shape

Later in the course, we shall look at *SIFT*, which is often more useful in practice as it considers both shape and appearance

Texture

The processing we have considered here uses shape only

Many more vision problems also involve appearance — that is clearly the case for *e.g.* face recognition

Sometimes, we have to identify similar textures (grass, sea, fabric...) so there need to be ways of describing them too

The lecture notes give the standard low-level ways of doing this, though they are non-examinable: Haralick's *grey-level co-occurrence matrices* and *Laws' masks* — although devised around 1980, there is still nothing around that really improves on them